# 1

# Introdução ao Design de Algoritmos

O que é um algoritmo? Um algoritmo é um procedimento para realizar uma tarefa específica. Um algoritmo é a ideia por trás de qualquer programa de computador razoável.

Para ser interessante, um algoritmo deve resolver um problema geral e bem especificado. Um problema algorítmico é especificado descrevendo o conjunto completo de instâncias nas quais ele deve trabalhar e de sua saída após a execução em uma dessas instâncias. Essa distinção, entre um problema e uma instância de um problema, é fundamental. Por exemplo, o problema algorítmico conhecido como classificação é definido da seguinte forma:

Problema: Classificação
Entrada: Uma sequência de n chaves a1,...,an.
Saída: A permutação (reordenação) da sequência de entrada de modo que a ···ÿa $_1$ ÿ um$_2$ ÿ $_{nÿ1}$ ÿ um$_n$.

Uma instância de classificação pode ser uma matriz de nomes, como {Mike, Bob, Sally, Jill, Jan}, ou uma lista de números como {154, 245, 568, 324, 654, 324}. Determinar que você está lidando com um problema geral é seu primeiro passo para resolvê-lo.

Um algoritmo é um procedimento que pega qualquer uma das instâncias de entrada possíveis e a transforma na saída desejada. Existem muitos algoritmos diferentes para resolver o problema de classificação. Por exemplo, a classificação por inserção é um método de classificação que começa com um único elemento (formando assim uma lista trivialmente classificada) e então insere incrementalmente os elementos restantes para que a lista permaneça classificada. Este algoritmo, implementado em C, é descrito abaixo:

SS Skiena, O Manual de Design de Algoritmos, 2ª ed., DOI: 10.1007/978-1-84800-070-4 1,

```
INSERÇÃO
INSERÇÃO
INSERÇÃO
EINSRTIONSORT
EINRSTIONSORT
EINRSTIONSORT
EIINRSTONSORT
EIINORSTNSORT
SORTIDO EIINNORS
DEPARTAMENTO DE EIINNORS
EIINNOORSTRT
EIINNOORRSSTT
EIINNOORRSSTT
```
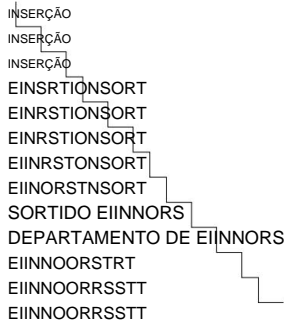
Figura 1.1: Animação da ordenação por inserção em ação (o tempo flui para baixo)

```
inserção_sort(item s[], int n) {

        int eu,j;                        /* contadores */

        para (i=1; i<n; i++) { j=i;

                enquanto ((j>0) && (s[j] < s[j-1])) { swap(&s[j],&s[j-1]);
                        j = j-1;

                }
        }
}
```

Uma animação do fluxo lógico deste algoritmo em uma instância particular (as letras na palavra “INSERTIONSORT”) é dada na Figura 1.1. Note a

generalidade deste algoritmo. Ele funciona tão bem em nomes quanto em números, dada a operação de comparação apropriada (<) para testar qual das duas chaves deve aparecer primeiro na ordem classificada. Pode ser prontamente verificado que este algoritmo ordena corretamente cada instância de entrada possível de acordo com nossa definição do problema de classificação.

Existem três propriedades desejáveis para um bom algoritmo. Buscamos algoritmos que sejam corretos e eficientes, ao mesmo tempo em que sejam fáceis de implementar. Essas metas podem não ser simultaneamente atingíveis. Em ambientes industriais, qualquer programa que pareça dar respostas boas o suficiente sem tornar o aplicativo mais lento é frequentemente aceitável, independentemente de existir um algoritmo melhor. A questão de encontrar a melhor resposta possível ou atingir a eficiência máxima geralmente surge na indústria somente após sérios problemas de desempenho ou legais.

Neste capítulo, focaremos nas questões de correção do algoritmo e adiaremos a discussão sobre as preocupações com a eficiência para o Capítulo 2. Raramente é óbvio se um dado
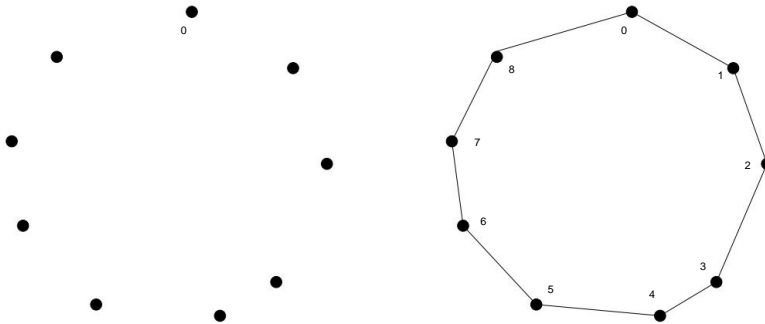
Figura 1.2: Uma boa instância para a heurística do vizinho mais próximo

algoritmo resolve corretamente um dado problema. Algoritmos corretos geralmente vêm com uma prova de correção, que é uma explicação do porquê sabemos que o algoritmo deve levar cada instância do problema ao resultado desejado. No entanto, antes de prosseguirmos, demonstramos por que "é óbvio" nunca é suficiente como uma prova de correção e geralmente está completamente errado.

## 1.1 Otimização do passeio do robô

Vamos considerar um problema que surge frequentemente em aplicações de fabricação, transporte e teste. Suponha que nos seja dado um braço robótico equipado com uma ferramenta, digamos um ferro de solda. Na fabricação de placas de circuito, todos os chips e outros componentes devem ser fixados no substrato. Mais especificamente, cada chip tem um conjunto de pontos de contato (ou fios) que devem ser soldados à placa. Para programar o braço robótico para este trabalho, devemos primeiro construir uma ordenação dos pontos de contato para que o robô visite (e solde) o primeiro ponto de contato, depois o segundo ponto, o terceiro e assim por diante até que o trabalho seja concluído. O braço robótico então retorna ao primeiro ponto de contato para se preparar para a próxima placa, transformando assim o caminho da ferramenta em um tour fechado, ou ciclo.

Robôs são dispositivos caros, então queremos o passeio que minimize o tempo que leva para montar a placa de circuito. Uma suposição razoável é que o braço do robô se move com velocidade fixa, então o tempo para viajar entre dois pontos é proporcional à distância deles. Em resumo, devemos resolver o seguinte problema de algoritmo:

Problema: Otimização do passeio do robô
Entrada: Um conjunto S de n pontos no plano.
Saída: Qual é o menor passeio de bicicleta que visita cada ponto no conjunto S?

Você recebeu a tarefa de programar o braço do robô. Pare agora mesmo e pense em um algoritmo para resolver esse problema. Ficarei feliz em esperar até que você encontre um. . .

Vários algoritmos podem vir à mente para resolver esse problema. Talvez a ideia mais popular seja a heurística do vizinho mais próximo. Começando de algum ponto p0, caminhamos primeiro para seu vizinho mais próximo p1. De p1, caminhamos para seu vizinho não visitado mais próximo, excluindo assim apenas p0 como candidato. Agora repetimos esse processo até ficarmos sem pontos não visitados, após o que retornamos a p0 para fechar o passeio. Escrita em pseudocódigo, a heurística do vizinho mais próximo se parece com isso:

```
Vizinho mais próximo (P)
    Escolha e visite um ponto inicial p0 de P
    p = p0
    eu = 0
    Enquanto ainda houver pontos não visitados i = i
        + 1
        Selecione pi para ser o ponto não visitado mais próximo de piÿ1
        Visite pi
    Retornar para p0 de pnÿ1
```

Este algoritmo tem muito a recomendar. É simples de entender e implementar. Faz sentido visitar pontos próximos antes de visitar pontos distantes para reduzir o tempo total de viagem. O algoritmo funciona perfeitamente no exemplo da Figura 1.2.
A regra do vizinho mais próximo é razoavelmente eficiente, pois analisa cada par de pontos (pi,pj ) no máximo duas vezes: uma vez ao adicionar pi ao passeio, a outra ao adicionar pj .
Contra todos esses pontos positivos, há apenas um problema. Este algoritmo está completamente errado.

Errado? Como pode estar errado? O algoritmo sempre encontra um passeio, mas não necessariamente encontra o passeio mais curto possível. Ele nem chega perto.
Considere o conjunto de pontos na Figura 1.3, todos os quais estão espaçados ao longo de uma linha. Os números descrevem a distância que cada ponto está à esquerda ou à direita do ponto rotulado '0'. Quando começamos do ponto '0' e caminhamos repetidamente até o vizinho não visitado mais próximo, podemos continuar pulando esquerda-direita-esquerda-direita sobre '0', pois o algoritmo não oferece nenhum conselho sobre como desempatar. Um passeio muito melhor (na verdade, ótimo) para esses pontos começa no ponto mais à esquerda e visita cada ponto enquanto caminhamos para a direita antes de retornar ao ponto mais à direita.

Agora tente imaginar a alegria da sua chefe ao assistir a uma demonstração do seu braço robótico saltando da esquerda para a direita, da esquerda para a direita durante a montagem de uma placa tão simples.

“Mas espere”, você pode estar dizendo. “O problema estava em começar no ponto '0'.
Em vez disso, por que não começamos a regra do vizinho mais próximo usando o ponto mais à esquerda como o ponto inicial p0? Ao fazer isso, encontraremos a solução ótima nesta instância.”

Isso é 100% verdade, pelo menos até girarmos nosso exemplo 90 graus. Agora todos os pontos estão igualmente mais à esquerda. Se o ponto '0' fosse movido apenas um pouco para a esquerda, ele seria escolhido como o ponto inicial. Agora o braço do robô vai pular para cima-baixo-cima-baixo em vez de esquerda-direita-esquerda-direita, mas o tempo de viagem será tão ruim quanto antes. Não importa o que você faça para escolher o primeiro ponto, a regra do vizinho mais próximo está fadada a funcionar incorretamente em certos conjuntos de pontos.
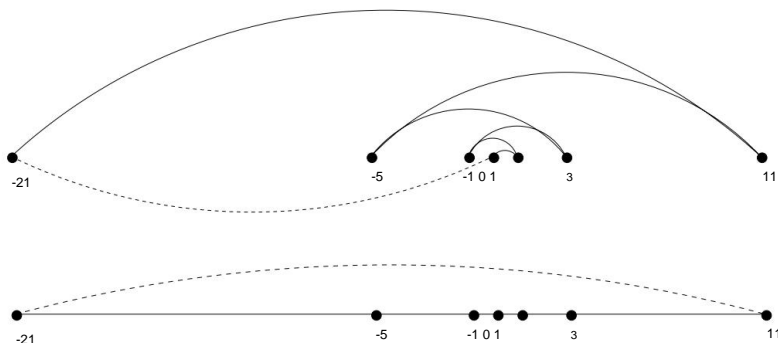
Figura 1.3: Uma instância ruim para a heurística do vizinho mais próximo, com a solução ótima

Talvez o que precisamos seja de uma abordagem diferente. Caminhando sempre para o mais próximo ponto é muito restritivo, pois parece nos prender a fazer movimentos que não fizemos quer. Uma ideia diferente pode ser conectar repetidamente o par mais próximo de pontos finais cuja conexão não criará um problema, como o término prematuro do ciclo. Cada vértice começa como sua própria cadeia de vértices. Depois de mesclar tudo juntos, terminaremos com uma única cadeia contendo todos os pontos nela. Conectar os dois pontos finais nos dá um ciclo. Em qualquer etapa durante a execução desta heurística de par mais próximo, teremos um conjunto de vértices simples e vértices disjuntos cadeias disponíveis para mesclar. Em pseudocódigo:

```
ParMaisPróximo(P)
    Seja n o número de pontos no conjunto P.
    Para i = 1 para n ÿ 1 faça
        d = ÿ
        Para cada par de pontos finais (s,t) de cadeias de vértices distintas
            se dist(s,t) ÿ d então sm = s, tm = t e d = dist(s,t)
        Conecte (sm,tm) por uma aresta
    Conecte os dois pontos finais por uma aresta
```

Esta regra do par mais próximo faz a coisa certa no exemplo da Figura 1.3. Ela começa conectando '0' aos seus vizinhos imediatos, os pontos 1 e ÿ1. Posteriormente, o par mais próximo alternará da esquerda para a direita, aumentando o caminho central em um link de cada vez uma vez. A heurística do par mais próximo é um pouco mais complicada e menos eficiente do que o anterior, mas pelo menos dá a resposta certa neste exemplo.

Mas isso não é verdade em todos os exemplos. Considere o que esse algoritmo faz no conjunto de pontos na Figura 1.4(l). Consiste em duas fileiras de pontos igualmente espaçados, com as linhas ligeiramente mais próximas (distância 1 ÿ e) do que os pontos vizinhos são espaçados dentro de cada linha (distância 1 + e). Assim, os pares de pontos mais próximos se estendem através da lacuna, não ao redor do limite. Depois de parear esses pontos, o mais próximo
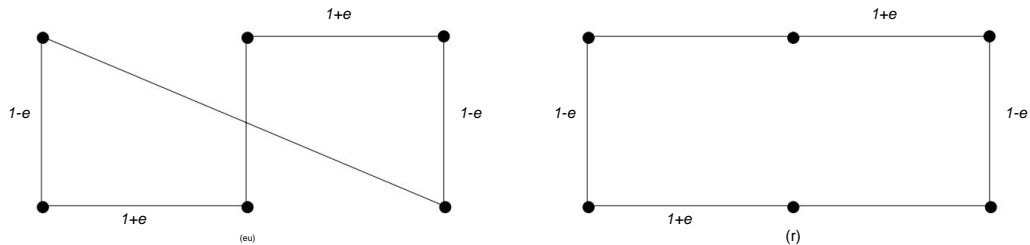
Figura 1.4: Uma instância ruim para a heurística do par mais próximo, com a solução ótima

os pares restantes conectarão esses pares alternadamente ao redor do limite. O comprimento total do caminho do passeio do par mais próximo é 3(1 ÿ e) + 2(1 + e) + (1 ÿ e)2 + (2 + 2e)2.
Comparado ao passeio mostrado na Figura 1.4(r), viajamos mais de 20% a mais do que o necessário quando e ÿ 0. Existem exemplos em que a penalidade é consideravelmente pior do que isso.

Portanto, este segundo algoritmo também está errado. Qual destes algoritmos tem melhor desempenho? Não dá para saber só olhando para eles. Claramente, ambas as heurísticas podem acabar com tours muito ruins em entradas de aparência muito inocente.

Neste ponto, você pode estar se perguntando como seria um algoritmo correto para o nosso problema. Bem, poderíamos tentar enumerar todas as ordenações possíveis do conjunto de pontos e, então, selecionar a ordenação que minimiza o comprimento total:

```
OptimalTSP(P) d = ÿ
    Para
    cada uma das n! permutações Pi do conjunto de pontos P
        Se (custo(Pi) ÿ d) então d = custo(Pi) e Pmin = Pi
    Retornar Pmin
```

Como todas as ordenações possíveis são consideradas, temos a garantia de terminar com o menor tour possível. Este algoritmo está correto, pois escolhemos a melhor de todas as possibilidades. Mas também é extremamente lento. O computador mais rápido do mundo não poderia esperar enumerar todas as 20! = 2.432.902.008.176.640.000 ordenações de 20 pontos em um dia. Para placas de circuito reais, onde n ÿ 1.000, esqueça.
Se todos os computadores do mundo trabalhassem em tempo integral, não conseguiriam resolver o problema antes do fim do universo, momento em que ele provavelmente se tornaria irrelevante.

A busca por um algoritmo eficiente para resolver esse problema, chamado de problema do caixeiro viajante (TSP), nos levará por boa parte deste livro. Se você precisa saber como a história termina, confira a entrada do catálogo para o problema do caixeiro viajante na Seção 16.4 (página 533).

Tarjan da Selva
O Problema dos Quatro Volumes
Árvore de Steiner
Processo encerrado
O Algorista do Presidente
Estado de Parada
Desafios de programação
Matemática "Discreta"
Apostas Calculadas
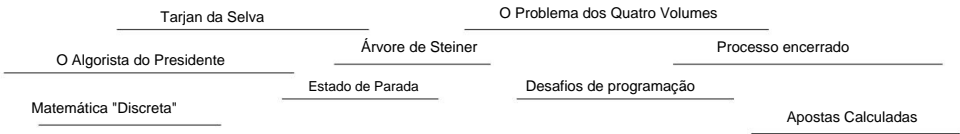
Figura 1.5: Uma instância do problema de agendamento de filmes não sobrepostos

Lição para levar para casa: Há uma diferença fundamental entre algoritmos, que sempre produzem um resultado correto, e heurísticas, que geralmente podem fazer um bom trabalho, mas não oferecem nenhuma garantia.

## 1.2 Selecionando os empregos certos

Agora considere o seguinte problema de agendamento. Imagine que você é um ator muito requisitado, que recebeu ofertas para estrelar n diferentes projetos de filmes em desenvolvimento. Cada oferta vem especificada com o primeiro e o último dia de filmagem. Para aceitar o trabalho, você deve se comprometer a estar disponível durante todo esse período. Portanto, você não pode aceitar simultaneamente dois trabalhos cujos intervalos se sobrepõem.

Para um artista como você, o critério para aceitação de trabalho é claro: você quer ganhar o máximo de dinheiro possível. Como cada um desses filmes paga a mesma taxa por filme, isso implica que você busca o maior conjunto possível de trabalhos (intervalos) de forma que nenhum deles entre em conflito um com o outro.

Por exemplo, considere os projetos disponíveis na Figura 1.5. Podemos estrelar no máximo quatro filmes, a saber, Matemática "Discreta", Desafios de Programação, Apostas Calculadas e um de Halting State ou Steiner's Tree.

Você (ou seu agente) deve resolver o seguinte problema de agendamento algorítmico:

Problema: Problema de agendamento de filmes
Entrada: Um conjunto I de n intervalos na linha.
Saída: Qual é o maior subconjunto de intervalos mutuamente não sobrepostos que podem ser selecionados de I?

Você recebeu a tarefa de desenvolver um algoritmo de agendamento para esta tarefa. Pare agora mesmo e tentar encontrar um. Novamente, ficarei feliz em esperar.

---

Há várias ideias que podem vir à mente. Uma é baseada na noção de que é melhor trabalhar sempre que houver trabalho disponível. Isso implica que você deve começar com o trabalho com a data de início mais próxima – afinal, não há outro trabalho em que você possa trabalhar, pelo menos durante o início deste período.

Figura 1.6: Instâncias ruins para as heurísticas (l) trabalho mais antigo primeiro e (r) trabalho mais curto primeiro.

PrimeiroTrabalhoPrimeiro(I)
Aceite o primeiro trabalho inicial j de I que não se sobreponha a nenhum trabalho aceito anteriormente e repita até que não haja mais trabalhos desse tipo.

Essa ideia faz sentido, pelo menos até percebermos que aceitar o primeiro emprego pode nos impedir de aceitar muitos outros empregos se esse primeiro emprego for longo. Confira a Figura 1.6(l), onde o épico “Guerra e Paz” é o primeiro emprego disponível e longo o suficiente para matar todos os outros prospectos.

Este mau exemplo naturalmente sugere outra ideia. O problema com “Guerra e Paz” é que ele é muito longo. Talvez devêssemos começar pegando o trabalho mais curto e continuar buscando o trabalho mais curto disponível a cada momento. Maximizar o número de trabalhos que fazemos em um determinado período está claramente conectado a executá-los o mais rápido possível. Isso produz a heurística:

TrabalhoMaisCurtoPrimeiro(I)
Enquanto (I = ÿ)
aceite o trabalho j mais curto possível de I.
Exclua j e qualquer intervalo que intercepte j de I.

Novamente, essa ideia faz sentido, pelo menos até percebermos que aceitar o trabalho mais curto pode nos bloquear de aceitar dois outros trabalhos, como mostrado na Figura 1.6(r). Embora a perda potencial aqui pareça menor do que com a heurística anterior, ela pode facilmente nos limitar à metade do retorno ótimo.

Neste ponto, um algoritmo em que tentamos todas as possibilidades pode começar a parecer bom, porque podemos ter certeza de que está correto. Se ignorarmos os detalhes de testar se um conjunto de intervalos é de fato disjunto, parece algo assim:

ExhaustiveScheduling(I) j = 0
Smax
= ÿ Para
cada um dos 2n subconjuntos Si de intervalos I Se (Si
for mutuamente não sobreposto) e (tamanho(Si) > j), então j =
tamanho(Si) e Smax = Si.
Retornar Smax

Mas quão lento é? A principal limitação é enumerar os 2n subconjuntos de n coisas. A boa notícia é que isso é muito melhor do que enumerar todos os n! pedidos

de n coisas, como proposto para o problema de otimização do passeio do robô. Há apenas cerca de um milhão de subconjuntos quando n = 20, que poderiam ser contados exaustivamente em segundos em um computador decente. No entanto, quando alimentado com n = 100 filmes, 2100 é muito, muito maior do que os 20! que fez nosso robô gritar "tio" no problema anterior.

A diferença entre nossos problemas de agendamento e robótica é que há um algoritmo que resolve o agendamento de filmes de forma correta e eficiente. Pense no primeiro trabalho a terminar — ou seja, o intervalo x que contém o ponto mais à direita que é o mais à esquerda entre todos os intervalos. Esse papel é desempenhado pela Matemática "Discreta" na Figura 1.5. Outros trabalhos podem muito bem ter começado antes de x, mas todos eles devem se sobrepor pelo menos parcialmente, para que possamos selecionar no máximo um do grupo. O primeiro desses trabalhos a terminar é x, então qualquer um dos trabalhos sobrepostos potencialmente bloqueia outras oportunidades à direita dele. Claramente, nunca podemos perder escolhendo x.
Isso sugere o seguinte algoritmo correto e eficiente:

```
Agendamento Ótimo(I)
    Enquanto (I = ÿ) faça
        Aceite o trabalho j de I com a data de conclusão mais próxima.
        Exclua j e qualquer intervalo que intercepte j de I.
```

Garantir a resposta ideal sobre todas as entradas possíveis é uma meta difícil, mas frequentemente atingível. Buscar contraexemplos que quebrem algoritmos fingidos é uma parte importante do processo de design de algoritmo. Algoritmos eficientes estão frequentemente à espreita por aí; este livro busca desenvolver suas habilidades para ajudá-lo a encontrá-los.

> Lição para levar para casa: Algoritmos aparentemente razoáveis podem facilmente estar incorretos. A correção do algoritmo é uma propriedade que deve ser cuidadosamente demonstrada.

## 1.3 Raciocínio sobre a correção

Espero que os exemplos anteriores tenham aberto seus olhos para as sutilezas da correção algorítmica. Precisamos de ferramentas para distinguir algoritmos corretos dos incorretos, o principal dos quais é chamado de prova.

Uma prova matemática adequada consiste em várias partes. Primeiro, há uma declaração clara e precisa do que você está tentando provar. Segundo, há um conjunto de suposições de coisas que são consideradas verdadeiras e, portanto, usadas como parte da prova.
Terceiro, há uma cadeia de raciocínio que leva você dessas suposições à declaração que você está tentando provar. Finalmente, há um pequeno quadrado ( ) ou QED na parte inferior para denotar █ue você terminou, representando a frase em latim para “assim é demonstrado”.

Este livro não vai enfatizar provas formais de correção, porque elas são muito difíceis de fazer direito e bastante enganosas quando você as faz errado. Uma prova é de fato uma demonstração. Provas são úteis somente quando são honestas; argumentos claros explicando por que um algoritmo satisfaz uma propriedade de correção não trivial.

Algoritmos corretos exigem exposição cuidadosa e esforços para mostrar tanto a correção quanto a incorreção. Desenvolvemos ferramentas para fazer isso nas subseções abaixo.

## 1.3.1 Expressando Algoritmos

Raciocinar sobre um algoritmo é impossível sem uma descrição cuidadosa da sequência de passos a serem realizados. As três formas mais comuns de notação algorítmica são (1) inglês, (2) pseudocódigo ou (3) uma linguagem de programação real.
Usaremos todos os três neste livro. Pseudocódigo é talvez o mais misterioso do grupo, mas é melhor definido como uma linguagem de programação que nunca reclama de erros de sintaxe. Todos os três métodos são úteis porque há uma compensação natural entre maior facilidade de expressão e precisão. Inglês é a linguagem de programação mais natural, mas menos precisa, enquanto Java e C/C++ são precisos, mas difíceis de escrever e entender. Pseudocódigo é geralmente útil porque representa um meio termo.

A escolha de qual notação é melhor depende de qual método você se sente mais confortável. Eu geralmente prefiro descrever as ideias de um algoritmo em inglês, mudando para um pseudocódigo mais formal, como uma linguagem de programação, ou mesmo código real para esclarecer detalhes suficientemente complicados.

Um erro comum que meus alunos cometem é usar pseudocódigo para disfarçar uma ideia mal definida para que pareça mais formal. Clareza deve ser o objetivo. Por exemplo, o algoritmo ExhaustiveScheduling na página 10 poderia ter sido melhor escrito em inglês como:

Agendamento Exaustivo (I)
Teste todos os $2^n$ subconjuntos de intervalos de I e retorne o maior subconjunto consistindo de intervalos mutuamente não sobrepostos.

> Lição para levar para casa: O coração de qualquer algoritmo é uma ideia. Se sua ideia não for revelada claramente quando você expressa um algoritmo, então você está usando uma notação de nível muito baixo para descrevê-la.

## 1.3.2 Problemas e Propriedades

Precisamos de mais do que apenas uma descrição de algoritmo para demonstrar correção. Precisamos também de uma descrição cuidadosa do problema que ele pretende resolver.

As especificações do problema têm duas partes: (1) o conjunto de instâncias de entrada permitidas e (2) as propriedades necessárias da saída do algoritmo. É impossível provar a correção de um algoritmo para um problema formulado de forma imprecisa. Em outras palavras, pergunte o problema errado e você obterá a resposta errada.

Algumas especificações de problemas permitem uma classe muito ampla de instâncias de entrada. Suponha que tivéssemos permitido que projetos de filmes em nosso problema de programação de filmes tivessem lacunas em

produção (ou seja, filmagens em setembro e novembro, mas um hiato em outubro).
Então, o cronograma associado a qualquer filme em particular consistiria em um conjunto dado de intervalos.
Nossa estrela estaria livre para assumir dois projetos intercalados, mas não sobrepostos (como o filme acima aninhado com uma filmagem em agosto e outubro).
O algoritmo de conclusão mais antigo não funcionaria para um problema de agendamento tão generalizado.
De fato, não existe algoritmo eficiente para esse problema generalizado.

Lição para levar para casa: Uma técnica importante e honrosa no design de algoritmos é estreitar o conjunto de instâncias permitidas até que haja um algoritmo correto e eficiente. Por exemplo, podemos restringir um problema de gráfico de gráficos gerais para árvores, ou um problema geométrico de duas dimensões para
um.

Há duas armadilhas comuns na especificação dos requisitos de saída de um problema.
Alguém está fazendo uma pergunta mal definida. Perguntar sobre a melhor rota entre dois lugares em um mapa é uma pergunta boba, a menos que você defina o que melhor significa. Você quer dizer a rota mais curta em distância total, ou a rota mais rápida, ou aquela que minimiza o número de curvas?

A segunda armadilha é criar objetivos compostos. Os três critérios de planejamento de caminho mencionados acima são todos objetivos bem definidos que levam a algoritmos de otimização corretos e eficientes. No entanto, você deve escolher um único critério. Um objetivo como Encontrar o caminho mais curto de a para b que não use mais do que o dobro de voltas necessárias é perfeitamente bem definido, mas complicado de raciocinar e resolver.

Eu o encorajo a verificar as declarações de problemas para cada um dos 75 problemas do catálogo na segunda parte deste livro. Encontrar a formulação correta para seu problema é uma parte importante da solução dele. E estudar a definição de todos esses problemas clássicos de algoritmos ajudará você a reconhecer quando outra pessoa pensou em problemas semelhantes antes de você.

### 1.3.3 Demonstrando Incorreção

A melhor maneira de provar que um algoritmo está incorreto é produzir uma instância na qual ele produz uma resposta incorreta. Tais instâncias são chamadas de contraexemplos.
Nenhuma pessoa racional jamais pulará em defesa de um algoritmo depois que um contraexemplo foi identificado. Instâncias muito simples podem matar instantaneamente heurísticas de aparência razoável com um toque rápido. Bons contraexemplos têm duas propriedades importantes:

- Verificabilidade – Para demonstrar que uma instância específica é um contra-exemplo para um algoritmo específico, você deve ser capaz de (1) calcular qual resposta seu algoritmo dará nesta instância e (2) exibir uma resposta melhor para provar que o algoritmo não a encontrou.

  Como você precisa manter o exemplo dado em sua cabeça para raciocinar sobre ele, uma parte importante da verificabilidade é...

- Simplicidade – Bons contra-exemplos eliminam todos os detalhes desnecessários. Eles deixam claro exatamente por que o algoritmo proposto falha. Uma vez que um contraexemplo tenha sido encontrado, vale a pena simplificá-lo até sua essência. Por exemplo, o contraexemplo da Figura 1.6(l) poderia ser simplificado e melhorado reduzindo o número de segmentos sobrepostos de quatro para dois.

Caçar contraexemplos é uma habilidade que vale a pena desenvolver. Ela tem alguma similaridade com a tarefa de desenvolver conjuntos de testes para programas de computador, mas depende mais de inspiração do que de exaustão. Aqui estão algumas técnicas para ajudar em sua busca:

- Pense pequeno – Observe que os contraexemplos de tour de robô que apresentei se resumiram a seis pontos ou menos, e os contraexemplos de agendamento a apenas três intervalos. Isso é indicativo do fato de que quando algoritmos falham, geralmente há um exemplo muito simples no qual eles falham. Algoristas amadores tendem a desenhar uma grande instância confusa e então encará-la desamparadamente. Os profissionais olham cuidadosamente para vários exemplos pequenos, porque são mais fáceis de verificar e raciocinar.

- Pense exaustivamente – Há apenas um pequeno número de possibilidades para o menor valor não trivial de n. Por exemplo, há apenas três maneiras interessantes de dois intervalos na linha poderem ocorrer: (1) como intervalos disjuntos, (2) como intervalos sobrepostos e (3) como intervalos adequadamente aninhados, um dentro do outro. Todos os casos de três intervalos (incluindo contraexemplos para ambas as heurísticas de filme) podem ser sistematicamente construídos adicionando um terceiro segmento em cada maneira possível para essas três instâncias.

- Caça à fraqueza – Se um algoritmo proposto for do tipo “sempre pegue o maior” (mais conhecido como algoritmo ganancioso), pense sobre por que isso pode provar ser a coisa errada a fazer. Em particular, . . .

- Vá para um empate – Uma maneira tortuosa de quebrar uma heurística gananciosa é fornecer instâncias onde tudo é do mesmo tamanho. De repente, a heurística não tem nada em que basear sua decisão e talvez tenha a liberdade de retornar algo subótimo como resposta.

- Procure extremos – Muitos contraexemplos são misturas de enorme e minúsculo, esquerda e direita, poucos e muitos, perto e longe. Geralmente é mais fácil verificar ou raciocinar sobre exemplos extremos do que sobre outros mais confusos. Considere duas nuvens de pontos bem agrupadas, separadas por uma distância d muito maior. O passeio TSP ideal será essencialmente 2d, independentemente do número de pontos, porque o que acontece dentro de cada nuvem não importa realmente.

Lição para levar para casa: procurar contra-exemplos é a melhor maneira de refutar a correção de uma heurística.

### 1.3.4 Indução e Recursão

A falha em encontrar um contraexemplo para um dado algoritmo não significa que "é óbvio" que o algoritmo está correto. Uma prova ou demonstração de correção é necessária. Muitas vezes a indução matemática é o método escolhido.

Quando aprendi sobre indução matemática pela primeira vez, parecia que i = n(n + 1)/2 completo uma fórmula como ou 2, então assumiu $\sum_{eu=1}^{e}$ para algum caso base como 1 mágica. Você provou que era verdadeira até n ÿ 1 antes de provar que era verdadeira para n geral usando a suposição. Isso foi uma prova? Ridículo!

Quando aprendi a técnica de programação de recursão, também parecia mágica completa. O programa testava se o argumento de entrada era algum caso base como 1 ou 2. Se não, você resolvia o caso maior dividindo-o em pedaços e chamando o próprio subprograma para resolver esses pedaços. Isso era um programa? Ridículo!

A razão pela qual ambas pareciam mágica é porque recursão é indução matemática. Em ambas, temos condições gerais e de contorno, com a condição geral quebrando o problema em pedaços cada vez menores. A condição inicial ou de contorno termina a recursão. Uma vez que você entenda recursão ou indução, você deve ser capaz de ver por que a outra também funciona.

Ouvi dizer que um cientista da computação é um matemático que só sabe provar coisas por indução. Isso é parcialmente verdade porque cientistas da computação são péssimos em provar coisas, mas principalmente porque muitos dos algoritmos que estudamos são recursivos ou incrementais.

Considere a correção da ordenação por inserção, que introduzimos no início deste capítulo. A razão pela qual está correto pode ser demonstrada indutivamente:

- O caso base consiste em um único elemento e, por definição, um caso de um único elemento a matriz está completamente classificada.

- Em geral, podemos assumir que os primeiros n ÿ 1 elementos da matriz A são compostos completamente classificado após n ÿ 1 iterações de classificação por inserção.

- Para inserir um último elemento x em A, encontramos onde ele vai, ou seja, o ponto único entre o maior elemento menor ou igual a x e o menor elemento maior que x. Isso é feito movendo todos os elementos maiores para trás em uma posição, criando espaço para x no local desejado.

∎

No entanto, é preciso desconfiar de provas indutivas, porque erros de raciocínio muito sutis podem surgir. Os primeiros são erros de limite. Por exemplo, nossa prova de correção de ordenação por inserção acima declarou corajosamente que havia um lugar único para inserir x entre dois elementos, quando nosso caso base era uma matriz de elemento único. É necessário maior cuidado para lidar adequadamente com os casos especiais de inserção dos elementos mínimos ou máximos.

A segunda e mais comum classe de erros de prova indutiva diz respeito a alegações de extensão cavallier. Adicionar um item extra a uma dada instância de problema pode fazer com que toda a solução ótima mude. Esse foi o caso em nosso problema de agendamento (veja Figura 1.7). O agendamento ótimo após inserir um novo segmento pode conter

Figura 1.7: Alterações em larga escala na solução ótima (caixas) após inserir um único intervalo (tracejado) na instância

nenhum dos segmentos de qualquer solução ótima particular antes da inserção. Ignorar corajosamente tais dificuldades pode levar a provas indutivas muito convincentes de algoritmos incorretos.

> Lição para casa: A indução matemática geralmente é a maneira correta de verificar a correção de um algoritmo de inserção recursivo ou incremental.

**Pare e pense: correção incremental**

Problema: Prove a correção do seguinte algoritmo recursivo para incrementar números naturais, ou seja, y ÿ y + 1:

```
Incremento(y)
        se y = 0 então retorne(1)
                senão se (y mod 2) = 1
                        então retorne(2 · Incremento(y/
                2)) senão retorne(y + 1)
```

---

Solução: A correção desse algoritmo certamente não é óbvia para mim. Mas como ele é recursivo e eu sou um cientista da computação, meu instinto natural é tentar prová-lo por indução.

O caso base de y = 0 é obviamente tratado corretamente. Claramente o valor 1 é retornado, e 0 + 1 = 1.

Agora, suponha que a função funcione corretamente para o caso geral de y = nÿ1. Dado isso, devemos demonstrar a verdade para o caso de y = n. Metade dos casos são fáceis, ou seja, os números pares (Para os quais (y mod 2) = 0), já que y + 1 é retornado explicitamente.

Para os números ímpares, a resposta depende do que é retornado por Increment(y/2). Aqui queremos usar nossa suposição indutiva, mas ela não está totalmente correta. Assumimos que increment funcionou corretamente para y = n ÿ 1, mas não para um valor que é cerca de metade dele. Podemos corrigir esse problema fortalecendo nossa suposição para declarar que o caso geral vale para todo y ÿ nÿ1. Isso não nos custa nada em princípio, mas é necessário para estabelecer a correção do algoritmo.

Agora, o caso de y ímpar (ou seja, y = 2m + 1 para algum inteiro m) pode ser tratado como:

$$2 \cdot \text{Incremento}((2m + 1)/2) = 2 \cdot \text{Incremento}(m + 1/2)$$
$$= 2 \cdot \text{Incremento}(m) = 2(m + 1) = 2m + 2 = y + 1$$

e o caso geral é resolvido. ■

## 1.3.5 Somas

Fórmulas de somas matemáticas surgem frequentemente na análise de algoritmos, que estudaremos no Capítulo 2. Além disso, provar a correção de fórmulas de somas é uma aplicação clássica da indução. Vários exercícios sobre provas indutivas de somas aparecem como exercícios no final deste capítulo. Para torná-los mais acessíveis, reviso os fundamentos das somas aqui.

As fórmulas de soma são expressões concisas que descrevem a adição de uma fórmula arbitrária. conjunto extraordinariamente grande de números, em particular a fórmula

$$\sum_{eu=1}^{e} f(i) = f(1) + f(2) + ... + f(n)$$

Existem formas fechadas simples para somas de muitas funções algébricas. Para por exemplo, já que n uns é n,

$$\sum_{eu=1}^{e} 1 = n$$

A soma dos primeiros n inteiros pode ser vista pareando o i-ésimo e o (n ÿ i + 1)-ésimo inteiros:

$$\sum_{eu=1}^{e} eu = \sum_{eu=1}^{n/2} (i + (n ÿ i + 1)) = n(n + 1)/2$$

Reconhecer duas classes básicas de fórmulas de soma levará você muito longe na análise de algoritmos:

- Progressões aritméticas – Já encontramos progressões aritméticas quando vimos S(n) = i = n(n + 1)/2 na análise de ordenação por seleção. não
  Da perspectiva do quadro geral, o importante é que a soma seja quadrática, não que a constante seja 1/2. Em geral,

$$S(n,p) = \sum_{eu}^{e} eu^{p} = ÿ(np+1)$$

Machine Translated by Google



para p ÿ 1. Assim, a soma dos quadrados é cúbica, e a soma dos cubos é quártica (se você usar tal palavra). A notação "big Theta" (ÿ(x)) será explicada adequadamente na Seção 2.2.

Para p < ÿ1, essa soma sempre converge para uma constante, mesmo quando n ÿ ÿ. O caso interessante é entre resultados em ...

- Série geométrica – Em progressões geométricas, o índice do loop afeta o expoente, ou seja,

$$G(n,a) = \sum_{eu=0}^{e} ai = a(an+1 ÿ 1)/(a ÿ 1)$$

A forma como interpretamos esta soma depende da base da progressão, ou seja, Quando a < 1, isso converge para uma constante mesmo quando n ÿ ÿ.

Essa convergência de séries prova ser o grande "almoço grátis" da análise de algoritmos. Isso significa que a soma de um número linear de coisas pode ser constante, não linear. Por exemplo, 1+1/2+1/4+1/8+... ÿ 2, não importa quantos termos somamos.

Quando a > 1, a soma cresce rapidamente com cada novo termo, como em 1 + 2 + 4 + 8 + 16 + 32 = 63. De fato, G(n,a) = ÿ(an+1) para a > 1.

**Pare e pense: Fórmulas fatoriais**

Problema: Prove que $\sum_{eu=1}^{e}$ i × i!=(n + 1)! ÿ 1 por indução.

---

Solução: O paradigma indutivo é direto. Primeiro verifique o caso base (aqui fazemos n = 1, embora n = 0 seria ainda mais geral):

$$\sum_{eu=1}^{1} eu \times eu! = 1 = (1 + 1)! ÿ 1=2 ÿ 1=1$$

Agora suponha que a afirmação seja verdadeira até n. Para provar o caso geral de n + 1, observe que a implementação do maior termo

$$\sum_{eu=1}^{n+1} eu \times eu! = (n + 1) \times (n + 1)! + \sum_{eu=1}^{n} eu \times eu!$$

revela o lado esquerdo da nossa suposição indutiva. Substituindo o lado direito dá nós

$$\sum_{eu=1}^{n+1} eu \times eu! = (n + 1) \times (n + 1)! + (n + 1)! ÿ 1$$

= (n + 1)! × ((n + 1) + 1) - 1 = (n + 2)! ÿ 1

Esse truque geral de separar o maior termo da soma para revelar uma instância da suposição indutiva está no cerne de todas essas provas.

■

# 1.4 Modelando o Problema

Modelagem é a arte de formular sua aplicação em termos de problemas precisamente descritos e bem compreendidos. Modelagem adequada é a chave para aplicar técnicas de design algorítmico a problemas do mundo real. De fato, modelagem adequada pode eliminar a necessidade de projetar ou mesmo implementar algoritmos, relacionando sua aplicação ao que foi feito antes. Modelagem adequada é a chave para usar efetivamente o "Guia do Mochileiro" na Parte II deste livro.

Aplicações do mundo real envolvem objetos do mundo real. Você pode estar trabalhando em um sistema para rotear tráfego em uma rede, para encontrar a melhor maneira de programar salas de aula em uma universidade ou para procurar padrões em um banco de dados corporativo. A maioria dos algoritmos, no entanto, é projetada para trabalhar em estruturas abstratas rigorosamente definidas, como permutações, gráficos e conjuntos. Para explorar a literatura de algoritmos, você deve aprender a descrever seu problema abstratamente, em termos de procedimentos em estruturas fundamentais.

## 1.4.1 Objetos Combinatórios

As chances são muito boas de que outros tenham tropeçado em seu problema algorítmico antes de você, talvez em contextos substancialmente diferentes. Mas para descobrir o que se sabe sobre seu "problema de otimização de widget" específico, você não pode esperar olhar em um livro em widget. Você deve formular a otimização de widget em termos de propriedades de computação de estruturas comuns, como:

- Permutações – que são arranjos, ou ordenações, de itens. Por exemplo, {1, 4, 3, 2} e {4, 3, 2, 1} são duas permutações distintas do mesmo conjunto de quatro inteiros. Já vimos permutações no problema de otimização de robôs e na classificação. Permutações são provavelmente o objeto em questão sempre que seu problema busca um "arranjo", "tour", "ordenação" ou "sequência".

- Subconjuntos – que representam seleções de um conjunto de itens. Por exemplo, {1, 3, 4} e {2} são dois subconjuntos distintos dos quatro primeiros inteiros. A ordem não importa em subconjuntos da mesma forma que importa com permutações, então os subconjuntos {1, 3, 4} e {4, 3, 1} seriam considerados idênticos. Vimos subconjuntos surgirem no problema de programação de filmes. Subconjuntos são provavelmente o objeto em questão sempre que seu problema busca um "cluster", "coleção", "comitê", "grupo", "embalagem" ou "seleção".
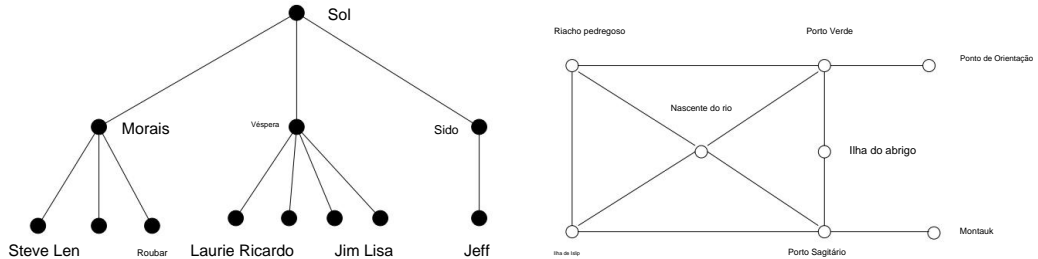
Figura 1.8: Modelagem de estruturas do mundo real com árvores e gráficos

- Árvores – que representam relacionamentos hierárquicos entre itens. A Figura 1.8(a) mostra parte da árvore genealógica do clã Skiena. Árvores são provavelmente o objeto em questão sempre que seu problema busca uma “hierarquia”, “relacionamento de dominância”, “relacionamento ancestral/descendente” ou “taxonomia”.

- Gráficos – que representam relacionamentos entre pares arbitrários de objetos. A Figura 1.8(b) modela uma rede de estradas como um gráfico, onde os vértices são cidades e as arestas são estradas conectando pares de cidades. Os gráficos são provavelmente o objeto em questão sempre que você busca uma “rede”, “circuito”, “teia” ou “relacionamento”.

- Pontos – que representam localizações em algum espaço geométrico. Por exemplo, as localizações dos restaurantes McDonald's podem ser descritas por pontos em um mapa/plano. Os pontos são provavelmente o objeto em questão sempre que seus problemas funcionam em “sites”, “posições”, “registros de dados” ou “localizações”.

- Polígonos – que representam regiões em alguns espaços geométricos. Por exemplo, as fronteiras de um país podem ser descritas por um polígono em um mapa/plano. Polígonos e poliedros são provavelmente o objeto em questão sempre que você estiver trabalhando com “formas”, “regiões”, “configurações” ou “limites”.

- Strings – que representam sequências de caracteres ou padrões. Por exemplo, os nomes dos alunos em uma classe podem ser representados por strings. Strings são provavelmente o objeto em questão sempre que você estiver lidando com “texto”, “caracteres”, “padrões” ou “rótulos”.

Todas essas estruturas fundamentais têm problemas de algoritmo associados, que são apresentados no catálogo da Parte II. A familiaridade com esses problemas é importante, porque eles fornecem a linguagem que usamos para modelar aplicativos. Para se tornar fluente nesse vocabulário, navegue pelo catálogo e estude as imagens de entrada e saída para cada problema. Entender esses problemas, mesmo em um nível de desenho animado/definição, permitirá que você saiba onde procurar mais tarde quando o problema surgir em seu aplicativo.
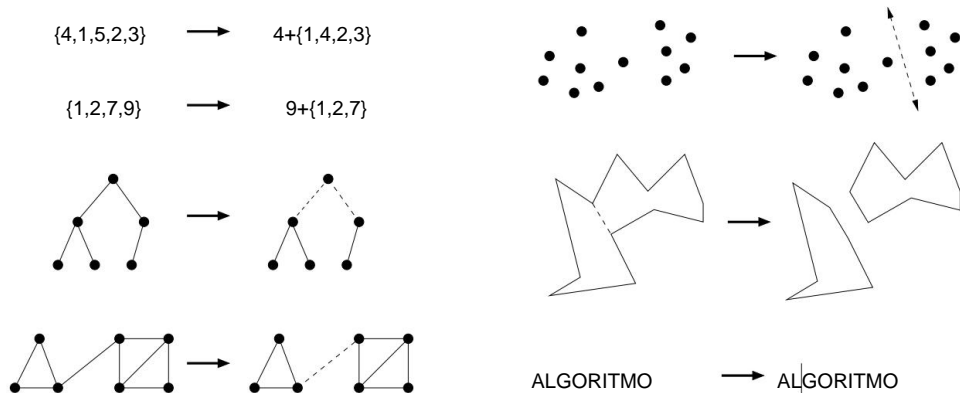
Figura 1.9: Decomposições recursivas de objetos combinatórios. (coluna da esquerda) Permutações, subconjuntos, árvores e gráficos. (coluna da direita) Conjuntos de pontos, polígonos e strings

Exemplos de modelagem de aplicativos bem-sucedidos serão apresentados nas histórias de guerra espaçadas ao longo deste livro. No entanto, algumas palavras de cautela são necessárias. O ato de modelar reduz seu aplicativo a um de um pequeno número de problemas e estruturas existentes. Tal processo é inerentemente restritivo, e certos detalhes podem não se encaixar facilmente no problema alvo fornecido. Além disso, certos problemas podem ser modelados de várias maneiras diferentes, algumas muito melhores do que outras.

A modelagem é apenas o primeiro passo no design de um algoritmo para um problema. Esteja alerta para como os detalhes de suas aplicações diferem de um modelo candidato, mas não seja muito rápido em dizer que seu problema é único e especial. Ignorar temporariamente detalhes que não se encaixam pode liberar a mente para se perguntar se eles realmente eram fundamentais em primeiro lugar.

Lição para levar para casa: modelar seu aplicativo em termos de estruturas e algoritmos bem definidos é o passo mais importante em direção a uma solução.

## 1.4.2 Objetos Recursivos

Aprender a pensar recursivamente é aprender a procurar coisas grandes que são feitas de coisas menores exatamente do mesmo tipo que a coisa grande. Se você pensa em casas como conjuntos de cômodos, então adicionar ou deletar um cômodo ainda deixa uma casa para trás.

Estruturas recursivas ocorrem em todo lugar no mundo algorítmico. De fato, cada uma das estruturas abstratas descritas acima pode ser pensada recursivamente. Você só precisa ver como pode quebrá-las, como mostrado na Figura 1.9:

- Permutações – Exclua o primeiro elemento de uma permutação de {1,...,n} coisas e você obtém uma permutação das n ÿ 1 coisas restantes. Permutações são objetos recursivos.

- Subconjuntos – Cada subconjunto dos elementos {1,...,n} contém um subconjunto de {1,...,n ÿ 1} tornado visível pela exclusão do elemento n se ele estiver presente. Subconjuntos são objetos recursivos.

- Árvores – Exclua a raiz de uma árvore e o que você obtém? Uma coleção de árvores menores. Exclua qualquer folha de uma árvore e o que você obtém? Uma árvore um pouco menor. Árvores são objetos recursivos.

- Gráficos – Exclua qualquer vértice de um gráfico e você obtém um gráfico menor. Agora divida os vértices de um gráfico em dois grupos, esquerdo e direito. Corte todas as arestas que se estendem da esquerda para a direita e o que você obtém? Dois gráficos menores e um monte de arestas quebradas. Os gráficos são objetos recursivos.

- Pontos – Pegue uma nuvem de pontos e separe-os em dois grupos desenhando uma linha. Agora você tem duas nuvens menores de pontos. Conjuntos de pontos são objetos recursivos.

- Polígonos – Inserir qualquer corda interna entre dois vértices não adjacentes de um polígono simples em n vértices o corta em dois polígonos menores. Polígonos são objetos recursivos.

- Strings – Exclua o primeiro caractere de uma string, e o que você obtém? Um string mais curta. Strings são objetos recursivos.

Descrições recursivas de objetos requerem regras de decomposição e casos base, ou seja, a especificação dos menores e mais simples objetos onde a decomposição para. Esses casos base são geralmente facilmente definidos. Permutações e subconjuntos de zero coisas presumivelmente se parecem com {}. A menor árvore ou gráfico interessante consiste em um único vértice, enquanto a menor nuvem de pontos interessante consiste em um único ponto. Polígonos são um pouco mais complicados; o menor polígono simples genuíno é um triângulo. Finalmente, a string vazia tem zero caracteres nela. A decisão de se o caso base contém zero ou um elemento é mais uma questão de gosto e conveniência do que qualquer princípio fundamental.

Essas decomposições recursivas definirão muitos dos algoritmos que usamos verá neste livro. Mantenha os olhos abertos para eles.

## 1.5 Sobre as histórias de guerra

A melhor maneira de aprender como o design cuidadoso de algoritmos pode ter um grande impacto no desempenho é olhar para estudos de caso do mundo real. Ao estudar cuidadosamente as experiências de outras pessoas, aprendemos como elas podem se aplicar ao nosso trabalho.

Espalhadas por todo este texto estão várias das minhas próprias histórias de guerra algorítmica, apresentando nossos esforços bem-sucedidos (e ocasionalmente malsucedidos) de design de algoritmos em aplicações reais. Espero que você consiga internalizar essas experiências para que elas sirvam como modelos para seus próprios ataques a problemas.

Cada uma das histórias de guerra é verdadeira. Claro, as histórias melhoram um pouco na recontagem, e o diálogo foi aprimorado para torná-las mais interessantes de ler. No entanto, tentei traçar honestamente o processo de ir de um problema bruto a uma solução, para que você possa assistir como esse processo se desenrolou.

O Oxford English Dictionary define um algorista como "alguém habilidoso em cálculos ou cálculos". Nessas histórias, tentei capturar um pouco da mentalidade do algorista em ação ao atacar um problema.

As várias histórias de guerra geralmente envolvem pelo menos um, e frequentemente vários, problemas do catálogo de problemas na Parte II. Eu faço referência à seção apropriada do catálogo quando tal problema ocorre. Isso enfatiza os benefícios de modelar sua aplicação em termos de problemas de algoritmo padrão. Ao usar o catálogo, você será capaz de extrair o que é conhecido sobre qualquer problema sempre que for necessário.

## 1.6 História de Guerra: Modelagem Psíquica

O telefonema chegou para mim do nada enquanto eu estava sentado no meu escritório.

"Professor Skiena, espero que você possa me ajudar. Sou o presidente da Lotto Systems Group Inc., e precisamos de um algoritmo para um problema que surgiu em nosso produto mais recente."

"Claro", respondi. Afinal, o reitor da minha escola de engenharia está sempre encorajando nosso corpo docente a interagir mais com a indústria.

"No Lotto Systems Group, comercializamos um programa projetado para melhorar a capacidade psíquica de nossos clientes de prever números vencedores da loteria.[1] Em uma loteria padrão, cada bilhete consiste em seis números selecionados de, digamos, 1 a 44. Assim, qualquer bilhete tem apenas uma chance muito pequena de ganhar. No entanto, após o treinamento adequado, nossos clientes podem visualizar, digamos, 15 números dos 44 e ter certeza de que pelo menos quatro deles estarão no bilhete vencedor. Você está comigo até agora?"

"Provavelmente não", respondi. Mas então me lembrei de como meu reitor nos encoraja a interagir com a indústria.

"Nosso problema é este. Depois que o vidente reduziu as escolhas para 15 números e tem certeza de que pelo menos 4 deles estarão no bilhete vencedor, precisamos encontrar a maneira mais eficiente de explorar essa informação. Suponha que um prêmio em dinheiro seja concedido sempre que você escolher pelo menos três dos números corretos em seu bilhete.
Precisamos de um algoritmo para construir o menor conjunto de bilhetes que devemos comprar para garantir que ganhemos pelo menos um prêmio."

"Supondo que o médium esteja correto?"

"Sim, assumindo que o médium esteja correto. Precisamos de um programa que imprima uma lista de todos os bilhetes que o médium deve comprar para minimizar seu investimento.
Você pode nos ajudar?"

Talvez eles tivessem habilidade psíquica, pois tinham vindo ao lugar certo. Identificar o melhor subconjunto de bilhetes para comprar era muito mais um algoritmo combinatório

[1] Sim, esta é uma história verdadeira.
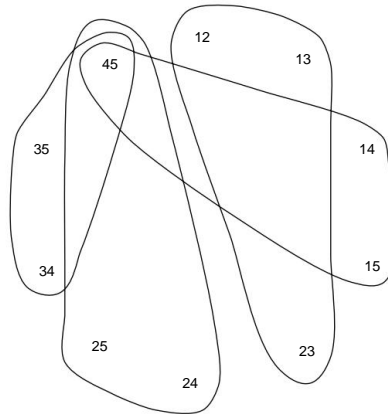
Figura 1.10: Cobrindo todos os pares de {1, 2, 3, 4, 5} com os bilhetes {1, 2, 3}, {1, 4, 5}, {2, 4, 5}, {3, 4, 5}

problema. Seria algum tipo de problema de cobertura, onde cada bilhete que comprássemos iria “cobrir” alguns dos possíveis subconjuntos de 4 elementos do conjunto do psíquico. Encontrar o menor conjunto absoluto de bilhetes para cobrir tudo era uma instância especial do conjunto de problemas NP-completo de cobertura (discutido na Seção 18.1 (página 621)), e presumivelmente computacionalmente intratável.

Foi de fato uma instância especial de cobertura de conjunto, completamente especificada por apenas quatro números: o tamanho n do conjunto candidato S (tipicamente n ÿ 15), o número de slots k para números em cada bilhete (tipicamente k ÿ 6), o número de números corretos psiquicamente prometidos j de S (digamos j = 4) e, finalmente, o número de números correspondentes l necessários para ganhar um prêmio (digamos l = 3). A Figura 1.10 ilustra uma cobertura de uma instância menor, onde n = 5, j = k = 3 e l = 2.

“Embora seja difícil encontrar o conjunto mínimo exato de bilhetes para comprar, com heurística eu deveria ser capaz de te deixar bem próximo do conjunto de bilhetes de cobertura mais barato”, eu disse a ele. “Isso será bom o suficiente?”

“Desde que gere conjuntos de tickets melhores do que o programa do meu concorrente, tudo bem. O sistema dele nem sempre garante uma vitória. Eu realmente aprecio sua ajuda nisso, Professor Skiena.”

“Uma última coisa. Se seu programa pode treinar pessoas para escolher ganhadores de loteria, por que você não o usa para ganhar na loteria você mesmo?”

“Espero falar com você novamente em breve, Professor Skiena. Obrigado pela ajuda.”

Desliguei o telefone e voltei a pensar. Parecia o projeto perfeito para dar a um aluno brilhante de graduação. Depois de modelá-lo em termos de conjuntos e subconjuntos, os componentes básicos de uma solução pareciam bastante diretos:

Machine Translated by Google



- Precisávamos da capacidade de gerar todos os subconjuntos de k números do conjunto candidato S. Algoritmos para gerar e classificar/desclassificar subconjuntos de conjuntos são apresentados na Seção 14.5 (página 452).

- Precisávamos da formulação correta do que significava ter um conjunto de cobertura de bilhetes comprados. O critério óbvio seria escolher um pequeno conjunto de bilhetes de modo que tivéssemos comprado pelo menos um bilhete contendo cada um dos l-subconjuntos de S e que poderiam pagar com o prêmio.

- Precisávamos manter o controle de quais combinações de prêmios cobrimos até agora. Buscamos bilhetes para cobrir o máximo possível de combinações de prêmios não cobertas até agora. As combinações cobertas atualmente são um subconjunto de todas as combinações possíveis. Estruturas de dados para subconjuntos são discutidas na Seção 12.5 (página 385). O melhor candidato parecia ser um vetor de bits, que responderia em tempo constante "essa combinação já está coberta?"

- Precisávamos de um mecanismo de busca para decidir qual bilhete comprar em seguida. Para tamanhos de conjunto suficientemente pequenos, poderíamos fazer uma busca exaustiva em todos os subconjuntos possíveis de bilhetes e escolher o menor. Para problemas maiores, um processo de busca aleatório como o simulation annealing (veja a Seção 7.5.3 (página 254)) selecionaria bilhetes para comprar para cobrir o máximo de combinações não cobertas possível. Ao repetir esse procedimento aleatório várias vezes e escolher a melhor solução, provavelmente chegaremos a um bom conjunto de bilhetes.

Excluindo os detalhes do mecanismo de busca, o pseudocódigo para o livro-mantendo algo parecido com isto:

```
Conjunto de Bilhetes de Loteria(n,k,l)
    Inicializar o e -elemento bit-vector V para todos falsos
    Enquanto houver uma entrada falsa em V
        Selecione um k-subconjunto T de {1,...,n} como o próximo bilhete a
        comprar Para cada um dos l-subconjuntos Ti de T, V [rank(Ti)] =
    true Relate o conjunto de bilhetes comprados
```

O brilhante estudante universitário, Fayyaz Younas, aceitou o desafio. Com base nessa estrutura, ele implementou um algoritmo de busca de força bruta e encontrou soluções ótimas para problemas com n ÿ 5 em um tempo razoável. Ele implementou um procedimento de busca aleatória para resolver problemas maiores, ajustando-o por um tempo antes de decidir pela melhor variante. Finalmente, chegou o dia em que poderíamos ligar para o Lotto Systems Group e anunciar que havíamos resolvido o problema.

"Nosso programa encontrou uma solução ótima para n = 15, k = 6, j = 6, l = 3, o que significava comprar 28 bilhetes."

"Vinte e oito bilhetes!" reclamou o presidente. "Você deve ter um vírus. Olha, esses cinco bilhetes vão dar para cobrir tudo duas vezes: {2, 4, 8, 10, 13, 14}, {4, 5, 7, 8, 12, 15}, {1, 2, 3, 6, 11, 13}, {3, 5, 6, 9, 10, 15}, {1, 7, 9, 11, 12, 14}."
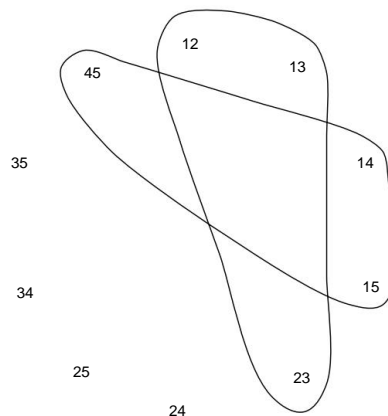
Figura 1.11: Garantindo um par vencedor de {1, 2, 3, 4, 5} usando apenas os bilhetes {1, 2, 3} e {1, 4, 5}

Nós brincamos com esse exemplo por um tempo antes de admitir que ele estava certo. Não havíamos modelado o problema corretamente! Na verdade, não precisávamos cobrir explicitamente todas as combinações vencedoras possíveis. A Figura 1.11 ilustra o princípio dando uma solução de dois bilhetes para nosso exemplo anterior de quatro bilhetes. Resultados pouco promissores como {2, 3, 4} e {3, 4, 5} concordam cada um em um par correspondente com os bilhetes da Figura 1.11. Estávamos tentando cobrir muitas combinações, e os médiuns mesquinhos não estavam dispostos a pagar por tamanha extravagância.

Felizmente, essa história tem um final feliz. O esboço geral da nossa solução baseada em busca ainda vale para o problema real. Tudo o que precisamos consertar é quais subconjuntos recebemos crédito por cobrir com um determinado conjunto de bilhetes. Após essa modificação, obtivemos o tipo de resultado que eles esperavam. O Lotto Systems Group aceitou com gratidão nosso programa para incorporar ao seu produto e, esperançosamente, ganhou o jackpot com ele.

A moral desta história é garantir que você modele o problema corretamente antes de tentar resolvê-lo. No nosso caso, criamos um modelo razoável, mas não trabalhamos duro o suficiente para validá-lo antes de começarmos a programar. Nossa má interpretação teria se tornado óbvia se tivéssemos trabalhado um pequeno exemplo manualmente e o tivéssemos passado para nosso patrocinador antes de começar a trabalhar. Nosso sucesso em nos recuperar desse erro é um tributo à correção básica de nossa formulação inicial e ao nosso uso de abstrações bem definidas para tarefas como (1) classificação/desclassificação de k-subconjuntos, (2) a estrutura de dados do conjunto e (3) busca combinatória.

## Notas do Capítulo

Todo livro de algoritmo decente reflete a filosofia de design de seu autor. Para estudantes que buscam apresentações e pontos de vista alternativos, recomendamos particularmente os livros de Corman, et. al [CLRS01], Kleinberg/Tardos [KT06] e Manber [Man89].

Provas formais de correção de algoritmo são importantes e merecem uma discussão mais completa do que somos capazes de fornecer neste capítulo. Veja Gries [Gri89] para uma introdução completa às técnicas de verificação de programa.

O problema de agendamento de filmes representa um caso muito especial do problema geral de conjunto independente, que é discutido na Seção 16.2 (página 528). A restrição limita as instâncias de entrada permitidas a grafos de intervalo, onde os vértices do grafo G podem ser representados por intervalos na linha e (i,j) é uma aresta de G se os intervalos se sobrepõem. Golumbic [Gol04] fornece um tratamento completo dessa classe interessante e importante de grafos.

As colunas Programming Pearls de Jon Bentley são provavelmente a coleção mais conhecida de "histórias de guerra" algorítmicas. Originalmente publicadas no Communications of the ACM, elas foram reunidas em dois livros [Ben90, Ben99]. The Mythical Man Month de Brooks [Bro95] é outra coleção maravilhosa de histórias de guerra, focada mais em engenharia de software do que em design de algoritmos, mas elas continuam sendo uma fonte de considerável sabedoria. Todo programador deveria ler todos esses livros, tanto por prazer quanto por percepção.

Nossa solução para o problema de cobertura do conjunto de bilhetes de loteria é apresentada com mais detalhes em [YS96].

## 1.7 Exercícios

Encontrando contraexemplos

1-1. [3] Mostre que a + b pode ser menor que min(a, b).

1-2. [3] Mostre que a × b pode ser menor que min(a, b).

1-3. [5] Projetar/desenhar uma rede rodoviária com dois pontos a e b de modo que a rota mais rápida entre a e b não é o caminho mais curto.

1-4. [5] Projete/desenhe uma rede rodoviária com dois pontos a e b de modo que a rota mais curta entre a e b não seja a rota com menos curvas.

1-5. [4] O problema da mochila é o seguinte: dado um conjunto de inteiros S = {s1, s2,...,sn}, e um número alvo T, encontre um subconjunto de S que some exatamente T. Por exemplo, existe um subconjunto dentro de S = {1, 2, 5, 9, 10} que some T = 22, mas não T = 23.

Encontre contraexemplos para cada um dos seguintes algoritmos para o problema da mochila. Ou seja, fornecer um S e T tal que o subconjunto seja selecionado usando o algoritmo não deixa a mochila completamente cheia, mesmo que tal solução exista.

(a) Coloque os elementos de S na mochila na ordem da esquerda para a direita, se eles couberem, ou seja, o algoritmo de primeiro ajuste.

(b) Coloque os elementos de S na mochila do menor para o maior, ou seja, o que melhor se ajusta algoritmo.

(c) Coloque os elementos de S na mochila do maior para o menor.

1-6. [5] O problema da cobertura do conjunto é o seguinte: dado um conjunto de subconjuntos $S_1, ..., S_m$ do conjunto universal U = {1, ..., n}, encontre o menor subconjunto de subconjuntos T ÿ S tal que ÿtiÿT ti = U. Por exemplo, existem os seguintes subconjuntos, $S_1$ = {1, 3, 5}, $S_2$ = {2, 4}, $S_3$ = {1, 4} e $S_4$ = {2, 5} A cobertura do conjunto seria então $S_1$ e $S_2$.

Encontre um contraexemplo para o seguinte algoritmo: Selecione o maior subconjunto para a cobertura e, em seguida, exclua todos os seus elementos do conjunto universal. Repita adicionando o subconjunto contendo o maior número de elementos descobertos até que todos estejam cobertos.

## Provas de Correção

1-7. [3] Prove a correção do seguinte algoritmo recursivo para multiplicar dois números naturais, para todas as constantes inteiras c ÿ 2.

```
função multiplicar(y,z)
    comentário Retorna o produto yz. se z
1.      = 0 então retorna(0) senão
2.      retorna(multiplicar(cy, z/c) + y · (z mod c))
```

1-8. [3] Prove a correção do seguinte algoritmo para avaliar um polinômio.
$P(x) = a_nx^n + a_{nÿ1}x^{nÿ1} + ... + a_1x + a_0$

```
função horner(A, x) p = An
    para i de
    n ÿ 1 a 0
        p = p ÿ x + Ai
    retornar p
```

1-9. [3] Prove a correção do seguinte algoritmo de classificação. função

```
bubblesort (A : list[1 ...n]) var int i, j

    para i de n para 1
        para j de 1 a i ÿ 1
            se (A[j] > A[j + 1])
                troque os valores de A[j] e A[j + 1]
```

## Indução

1-10. [3] Prove que $\sum_{eu=1}^{e}$ i=n(n + 1)/2 para n ÿ 0, por indução. 1-11. [3] Prove que $\sum_{eu=1}^{e}$ i 2=n(n + 1)(2n + 1)/6 para n ÿ 0, por indução. 1-12. [3] Prove que $\sum_{eu=1}^{e}$ i 3=n2(n + 1)2/4 para n ÿ 0, por indução. 1-13. [3] Prove que

$$\sum_{eu=1}^{e} eu(i + 1)(i + 2) = n(n + 1)(n + 2)(n + 3)/4$$

1-14. [5] Prove por indução em n ÿ 1 que para cada a = 1,

$$\overset{e}{\underset{eu=0}{}} ai = \frac{um+1 ÿ 1}{um - 1}$$

1-15. [3] Prove por indução que para n ÿ 1,

$$\overset{e}{\underset{eu=1}{}} \frac{1}{eu(i + 1)} = \frac{e}{n + 1}$$

1-16. [3] Prove por indução que n3 + 2n é divisível por 3 para todo n ÿ 0.

1-17. [3] Prove por indução que uma árvore com n vértices tem exatamente n ÿ 1 arestas.

1-18. [3] Prove por indução matemática que a soma dos cubos dos primeiros n inteiros positivos é igual ao quadrado da soma desses inteiros, ou seja

$$\overset{e}{\underset{eu=1}{}} eu^{3} = (n \underset{eu=1}{} eu)^{2}$$

## Estimativa

1-19. [3] Todos os livros que você possui totalizam pelo menos um milhão de páginas? Quantas páginas totais estão armazenadas na biblioteca da sua escola?

1-20. [3] Quantas palavras há neste livro didático?

1-21. [3] Quantas horas são um milhão de segundos? Quantos dias? Responda a estas perguntas perguntas fazendo todas as contas de cabeça.

1-22. [3] Estime quantas cidades e vilas existem nos Estados Unidos.

1-23. [3] Estime quantas milhas cúbicas de água fluem da foz do Rio Mississippi a cada dia. Não procure nenhum fato suplementar. Descreva todas as suposições que você fez para chegar à sua resposta.

1-24. [3] O tempo de acesso à unidade de disco é normalmente medido em milissegundos (milésimos de segundo) ou microssegundos (milionésimos de segundo)? Sua memória RAM acessa uma palavra em mais ou menos de um microssegundo? Quantas instruções sua CPU pode executar em um ano se a máquina for deixada funcionando o tempo todo?

1-25. [4] Um algoritmo de classificação leva 1 segundo para classificar 1.000 itens em sua máquina local. Quanto tempo levará para classificar 10.000 itens...

(a) se você acredita que o algoritmo leva um tempo proporcional a n2, e (b) se você

acredita que o algoritmo leva um tempo aproximadamente proporcional a n log n?

Projetos de Implementação 1-26.

[5] Implemente as duas heurísticas TSP da Seção 1.1 (página 5). Qual delas fornece soluções de melhor qualidade na prática? Você pode elaborar uma heurística que funcione melhor do que ambas?

1-27. [5] Descreva como testar se um dado conjunto de bilhetes estabelece cobertura suficiente no problema da loteria da Seção 1.6 (página 23). Escreva um programa para encontrar bons conjuntos de bilhetes.

Problemas de entrevista

1-28. [5] Escreva uma função para executar a divisão inteira sem usar os operadores / ou *. Encontre uma maneira rápida de fazer isso.

1-29. [5] Existem 25 cavalos. No máximo, 5 cavalos podem correr juntos ao mesmo tempo. Você deve determinar o cavalo mais rápido, o segundo mais rápido e o terceiro mais rápido. Encontre o número mínimo de corridas em que isso pode ser feito.

1-30. [3] Quantos afinadores de piano existem no mundo inteiro?

1-31. [3] Quantos postos de gasolina existem nos Estados Unidos?

1-32. [3] Quanto pesa o gelo em uma pista de hóquei?

1-33. [3] Quantas milhas de estradas existem nos Estados Unidos?

1-34. [3] Em média, quantas vezes você teria que abrir o telefone de Manhattan livro aleatoriamente para encontrar um nome específico?

**Desafios de programação**

Esses problemas de desafio de programação com julgamento de robôs estão disponíveis em http://www.programming-challenges.com ou http://online-judge.uva.es.

1-1. “O Problema 3n + 1” – Desafios de Programação 110101, UVA Judge 100.

1-2. “A Viagem” – Desafios de Programação 110103, UVA Judge 10137.

1-3. “Votação Australiana” – Desafios de Programação 110108, Juiz UVA 10142.